УДК 004.451.2, 004.82, 004.021, 519.6, 004.42

Операционная семантика операторов передачи управления в языке C на языке ABML

Ануреев И.С. (Институт систем информатики СО РАН)

В работе рассматривается онтологический подход к заданию операционной семантики операторов передачи управления языка программирования C. В качестве формального средства используется предметно-ориентированный язык ABML, ранее предложенный для спецификации дискретных динамических систем, ориентированных на знания, структурированные в онтологиях. Показано, что операционную семантику фрагментов языков программирования, заданную в терминах систем переходов, можно интерпретировать как динамическую систему и формализовать средствами ABML.

В статье вводится онтология операторов передачи управления языка C, включающая операторы goto, break, continue и return, а также онтологии конструкций, реагирующих на передачу управления, таких как помеченные операторы, блоки и оператор switch. Для этих онтологических моделей задается операционная семантика в виде атрибутных замыканий, вычисляемых относительно агентов и окружения.

Особое внимание уделяется адаптации языка ABML к задачам задания операционной семантики, включая уточнение понятия атрибутного замыкания, введение стадий вычисления и явное моделирование контекста выполнения. Предложенный подход обеспечивает модульность, расширяемость и наглядность спецификации семантики.

Полученные результаты демонстрируют применимость онтологического моделирования для формального описания семантики языков программирования и создают основу для дальнейшего расширения подхода на другие конструкции языка C, а также на анализ и верификацию программ.

Ключевые слова: операционные семантики, онтологии языков программирования, модели языков программирования, атрибутные замыкания, ABML, операторы передачи управления

1. Введение

Формальное задание семантики языков программирования является одной из ключевых задач теории программирования и формальных методов. Операционная семантика, описывающая поведение программ через последовательность элементарных шагов выполнения, традиционно задается с помощью систем переходов, абстрактных машин или пра-

вил вывода. Такие описания, хотя и обладают высокой точностью, часто оказываются слабо структурированными, трудно расширяемыми и плохо приспособленными для повторного использования при анализе различных фрагментов языка.

В последние годы заметный интерес вызывает применение онтологического подхода к моделированию программных систем. Онтологии позволяют явно фиксировать структуру предметной области, типы сущностей и отношения между ними, что делает модели более прозрачными и пригодными для автоматизированной обработки. В контексте языков программирования это открывает возможность представлять синтаксические и семантические конструкции языка в виде онтологических моделей, а правила их функционирования – в виде формализованных механизмов изменения знаний.

В работе [63] был предложен язык ABML (Attribute-Based Modeling Language), предназначенный для спецификации и прототипирования дискретных динамических систем, ориентированных на знания, структурированные в онтологиях. Язык ABML объединяет онтологическое моделирование с элементами функционального и процедурного программирования и предоставляет средства для задания объектов, атрибутов, типов, сопоставления с образцом и атрибутных замыканий. В предыдущей работе было показано, что ABML может эффективно использоваться для моделирования динамики реальных технических и информационных систем.

Настоящая статья развивает данный подход и рассматривает возможность применения ABML для задания операционной семантики языков программирования. Основная идея заключается в том, что систему переходов, лежащую в основе операционной семантики, можно рассматривать как дискретную динамическую систему, а значит, описывать ее средствами ABML. Однако специфика языков программирования – наличие контекста выполнения, стеков, областей видимости, передачи управления – требует адаптации базовых механизмов языка.

В качестве предмета исследования выбран фрагмент языка C, связанный с операторами передачи управления. Эти операторы играют важную роль в управлении потоком выполнения программы и существенно усложняют формальное описание семантики из-за нелокальных переходов, взаимодействия с блоками, циклами и оператором switch. В статье предлагается онтологическая модель операторов передачи управления и связанных с ними конструкций, а также формальное задание их операционной семантики на языке ABML.

Целью работы является демонстрация того, что онтологический подход в сочетании с языком ABML позволяет получить модульное, расширяемое и формально точное описание операционной семантики операторов передачи управления языка C. Полученные результаты могут служить основой для дальнейшего расширения модели, а также для исследований в области анализа, верификации и интерпретации программ.

Статья имеет следующую структуру. В разделе 2 рассматривается адаптация языка ABML к задачам задания операционной семантики языков программирования и вводятся необходимые уточнения базовых понятий. В разделе 3 описываются этапы построения операционной семантики в терминах ABML. Раздел 4 посвящен онтологии операторов передачи управления языка C, а в разделе 5 вводится онтология конструкций, связанных с передачей управления. В разделе 6 описываются модели агентов и окружения, используемые для задания контекста выполнения. Разделы 7 и 8 содержат формальное описание операционной семантики операторов передачи управления и связанных с ними конструкций. В разделе 9 проведен анализ родственных работ. В заключении подводятся итоги работы и обсуждаются направления дальнейших исследований.

2. Адаптация ABML для разработки операционных семантик

В работе [63] был предложен предметно-ориентированный язык ABML, предназначенный для спецификации и прототипирования дискретных динамических систем, ориентированных на знания, структурированные в онтологиях. Основная идея применения этого языка для спецификации операционной семантики языков программирования заключается в следующем. Если задавать операционную семантику языков программирования с помощью систем переходов, то такие системы переходов можно рассматривать как дискретные динамические системы и, таким образом, можно применить язык ABML для спецификации таких систем.

Однако эти системы имеют свои особенности, и чтобы учитывать их, требуется некоторые модификации языка ABML. Опишем ниже эти модификации.

Понятие атрибутного замыкания переопределяется следующим образом. Константный объект ac называется *атрибутным замыканием* относительно атрибута a и типа t, если выполняются следующие условия:

- (aget ac "attribute") = a;
- (aget ac "instance") = i, где i является экземпляром типа t;

- `"agent"` – экземпляр типа изменяемых объектов `"agent"`. Агенты выполняют двоякую роль. С одной стороны, агенты хранят в своих атрибутах знания, необходимые для выполнения конструкций языка программирования (например, значения переменных, типы переменных, распределение памяти и т. п.). С другой стороны, агентов можно рассматривать как отдельных исполнителей, специфика которых определяется хранимым в них знанием (например, для определения потоков, процессов и т. п.). Обязательным атрибутом агентов является атрибут `"value"`, хранящий последнее вычисленное в этом агенте значение;
- `"env"` – экземпляр типа изменяемых объектов `"env"`. Окружение во-первых, хранит информацию, общую для всех агентов, а во-вторых, позволяет агентам обмениваться этой информацией через атрибуты окружения. В окружении также реализуется обобщение концепции стека, широко используемой в языках программирования для откладывания вычислений (например, элементом стека может быть функция с аргументами из ее текущего вызова). Реализация основана на двух обязательных атрибутах окружения. Атрибут `"agents"` типа `(listt "agent")` хранит список всех действующих агентов. Атрибут `"aclosures"` типа `(cot :amap "agent" (listt cot))` моделирует стек для каждого агента, хранящий отложенные атрибутные замыкания для этого агента. Декларации типов для окружения и агентов задаются пользователем отдельно для каждого языка программирования.

Остальные атрибуты объекта ac образуют контекст вычисления атрибута a. Также в контекст вычисления добавляется обязательный атрибут `"stage"`, значения которого характеризуют отдельные стадии вычисления атрибутного замыкания. Конкретный набор этих стадий зависят от типа экземпляра, для которого вычисляется атрибут в замыкании ac. Например, для условного оператора можно выделить 3 стадии: вычисление условия, выбор ветви и вычисление выбранной ветви. Использование данного атрибута способствует модульности операционной семантики, а также в совокупности с механизмом стеков, описанным ниже, упрощает обработку исключительных ситуаций (появляющихся как результат выполнения операторов передачи управления, операторов порождения исключений и т. п.), которые могут встретиться на каждой стадии.

Таким образом, к обязательным атрибутам атрибутных замыканий `"attribute"` и `"instance"` добавляются еще два атрибута `"agent"` и `"env"`. В частности, эти атрибуты также будут сохраняться при выполнении функции `(clear-aclosure` ac`)`, которая удаля-

ет атрибуты из контекста вычисления.

В язык ABML добавляются следующие функции работы со стеками окружения:

- push-aclosure – добавляет атрибутное замыкание *ac* в стек соответствующего агента, связанного с *ac*. Она имеет следующую семантику:

```
(defun push-aclosure (ac)
  (match :ap ac "agent" a :ap ac "env" e
    :ap e "agents" al
    :ap e (aseq "aclosures" a) cl
    :do (aset e "aclosures" a (cons ac cl))
    :v (not (member a al)) T
    :do (aset e "agents" (cons a al))))
```

- (pop-aclosure *ac*) – удаляет атрибутное замыкание *ac* из стека соответствующего агента, возвращая его в качестве значения этой функции. Она имеет следующую семантику:

```
(defun pop-aclosure (ac)
  (match :ap ac "env" e :ap e "agents" al
    :v (not (null al)) T :exit nil
    :p (car al) a :ap e (aseq "aclosures" a) cl
    :v (not (null cl)) T :exit nil
    :do (aset e "aclosures" a (cdr cl)) (car cl)))
```

- (peek-aclosure *ac*) – читает атрибутное замыкание *ac* из стека соответствующего агента, возвращая его в качестве значения этой функции. Она имеет следующую семантику:

```
(defun peek-aclosure (ac)
  (match :ap ac "env" e :ap e "agents" al
    :v (not (null al)) T :exit nil
    :p (car al) a :ap e (aseq "aclosures" a) cl
    :v (not (null cl)) T :exit nil
    :do (car cl)))
```

Имеются также сокращения

```
(update-push-aclosure ac ...)
clear-update-push-aclosure ac ...)
```

для часто используемых операций

```
(push-aclosure (aset ac ...))
(update-push-aclosure (clear-aclosure ac))
```

В префикс декларации атрибутного замыкания добавляется элемент `:value` p, который связывает с переменной p значение атрибута `"value"` агента `(aget` ac `"agent")`, связанного с атрибутным замыканием ac. Таким образом, декларация атрибутного замыкания принимает один из следующих видов:

```
(aclosure ac :attribute a :type t :instance i :value p s1 s2 s3
  :do e1 ... er)
(aclosure ac :attribute a :type t :instance i :value p s1 s2 s3
  :match c1 ... cr)
(aclosure ac :attribute a :type t :instance i :value p s1 s2 s3
  :nmatch c1 ... cr)
```

где s_1, s_2 и s_3 имеют вид

```
:a1 p1 ... :an pn
:ap w1 b1 q1 ... :ap wm bm qm
:p u1 t1 ... :p uk tk
```

соответственно.

Результат вычисления атрибутного замыкания ac для атрибута a и типа t определяется λ-функцией `(lambda (`ac`)` b`)`, где тело b имеет вид

```
(match :ap ac "instance" i :ap ac (aseq "agent" "value") p
  :ap ac a1 p1 ... :ap ac an pn :ap w1 b1 q1 ... :ap wm bm qm
  :p u1 t1 ... :p uk tk :do e1 ... er (next-aclosure ac))

(match :ap ac "instance" i :ap ac (aseq "agent" "value") p
  :ap ac a1 p1 ... :ap ac an pn :ap w1 b1 q1 ... :ap wm bm qm
  :p u1 t1 ... :p uk tk c1 ... cr (next-aclosure ac))
```

```
8
9  (match :ap ac "instance" i :ap ac (aseq "agent" "value") p
10   :ap ac a1 p1 ... :ap ac an pn :ap w1 b1 q1 ... :ap wm bm qm
11   :p u1 t1 ... :p uk tk (nmatch c1 ... cr) (next-aclosure ac))
```

соответственно. Здесь выражения e_1, ..., e_r, c_1, ..., c_r могут зависеть от параметров i, ac, p, p_1, ..., p_n, q_1, ..., q_m, t_1, ..., t_k.

По-прежнему элементы префикса декларации атрибутного замыкания могут как переставляться, так и опускаться.

Функция (next-aclosure ac) определяет какое атрибутное замыкание требуется взять из стеков окружения и выполнить после того, как завершится выполнение замыкания ac. Эта функция определяется пользователем в зависимости от специфики языка программирования, для которого строится операционная семантика. В этой статье мы используем следующее определение:

```
1  (defun next-aclosure (ac)
2    (match :ap ac "value" v
3      :v (not (is-instance v "stop next aclosure")) T :exit v
4      :ap ac "agent" a :ap ac "env" e :ap e (aseq "aclosures" a) st
5      :v (null st) T :exit (eval-aclosure (car st))
6      :ap e "agents" al
7      :v (not (null al)) T :exit nil
8      :p (car al) a1 :ap e (aseq "aclosures" a1) st1
9      :v (null st1) T :exit (aset e "aclosures" a (cdr cl))
10       (eval-aclosure (car st1))
11     :do (aset e "agents" a (cdr al)) next-aclosure (ac)))
```

Оно выбирает первый элемент стека агента, связанного с замыканием ac, а если стек пуст, то первый элемент первого агента из списка агентов в окружении, для которого стек не пуст. Если стеки для всех агентов пусты, то эта функция ничего не делает.

Для моделирования ситуации, когда next-aclosure ничего не делает, и, таким образом, ABML-программа завершает свою работу, используется значение типа "stop next aclosure" из атрибута "value" агента, связанного с замыканием ac. Этот тип определяется следующим образом:

```
(cot "stop next aclosure" :at "type" string :at "aclosure" (cot))
```

Атрибут "type" хранит тип остановки программы (например, "error"), а атрибут "aclosure" хранит замыкание, на котором произошла остановка.

3. Этапы построения операционной семантики на языке ABML

Подход к построению операционной семантики фрагмента языка программирования на языке ABML состоит из следующих шагов:

1. Построить онтологию фрагмента языка программирования как набор типов объектов языка ABML, соответствующих синтаксическим и семантическим конструкциям этого фрагмента.
2. Определить типы для агентов и окружения.
3. Задать операционную семантику фрагмента как набор атрибутных замыканий относительно атрибута "opsem" и всех типов, определенных на шаге 1, которые соответствуют исполняемым конструкциям фрагмента. В этом случае, вычисление атрибута "opsem" для онтологической модели исполнимой конструкции через атрибутное замыкание соответствует вычислению операционной семантики этой конструкции относительно агента, окружения и других параметров, задаваемых этим атрибутным замыканием.

4. Онтология операторов передачи управления

Онтология операторов передачи управления задается следующим набором типов:

```
(typedef "jump statement" (uniont
  "goto1" "continue" "break" "return1" "return1"))
(mot "goto1" :at 1 "identifier")
(mot "continue")
(mot "break")
(mot "return")
(mot "return1" :at 1 "expression")
```

Индексы 1, 2, 3 и т. д. в именах типов показывают позиции аргументов. Типы "continue", "break" и "return", соответствующие операторам `continue`, `break` и `return` без аргументов, не имеют атрибутов. Атрибутом 1 типа "goto" является метка оператора `goto`, а

атрибутом 1 типа "return1" является выражение, связанное с оператором return. Тип "jump statement" моделирует все виды операторов передачи управления и определяется как их объединение.

5. Онтология операторов, связанных с операторами передачи управления

Определим также операторы и выражения, которые реагируют на передачу управления, разбив их на группы.

Первую группу операторов составляют помеченные операторы. Для упрощения операционной семантики мы рассматриваем в качестве помеченных операторов различные виды меток без следующих за ними операторов. Эта группа операторов моделируется следующими типами:

```
(typedef "labeled statement" (uniont
  "label1" "case1" "default"))
(mot "label1" :at 1 "label name")
(mot "case1" :at 1 "constant expression")
(mot "default")
```

Типы "label1" и "case1", соответствующие обычных меткам и case-меткам, имеют один атрибут 1 со значениями типов "label name" и "constant expression", представляющих метки и константные выражения, соответственно. Тип "default" моделирует default-метки.

Вторую группу составляют операторы блока. Их модели определяются следующим образом:

```
(mot "{1}"
  :at 1 (listt (uniont "declaration" "statement"))
  :at "variables" (listt "variable")
  :at "label position" (cot :amap "label name" nat))
```

Атрибут "variables" хранит список переменных, объявляемых на верхнем уровне в списке операторов блока. Это знание нужно для того, чтобы при переходе к телу этого оператора для переменных с теми же именами, как переменные из этого списка, сохранять ячейки памяти, связанные с ними, поскольку эти переменные будут прятаться при объявлениях

соответствующих переменных внутри тела, а при выходе из тела цикла восстанавливать старые связи между переменными и ячейками.

Атрибут "variable location" в этом типе хранит отображение меток, встречающихся в списке операторов блока, в их позиции в этом списке.

Третью группу образуют онтологические модели операторов `switch`, представленные типом "switch(1)2":

```
(mot "switch(1)2" :at 1 "expression" :at 2 (listt "statement")
  :at "variables" (listt "variable"))
```

Атрибуты 1 и 2 имеют типы "expression" и (listt "statement") и задают управляющее выражение и тело этих операторов.

Четвертую группу образуют операторы итерации, включающих операторы `while`, операторы `do-while` и два вида операторов `for` без и с декларацией переменных. Эта группа моделируется следующими типами:

```
(typedef "iteration statement" (uniont "while(1)2"
  "do1while(2)" "for(1;2;3)4" "for(var1;2;3)4"))

(mot "while(1)2" :at 1 "expression" :at 2 "statement")

(mot "do1while(2)" :at 1 "statement" :at 2 "expression")

(mot "for(1;2;3)4" :at 1 "expression" :at 2 "expression"
  :at 3 "expression" :at 4 "statement")

(mot "for(var1;2;3)4" :at 1 "declaration" :at 2 "expression"
  :at 3 "expression" :at 4 "statement"
  :at "variables" (listt "variable"))
```

Атрибут "variables" в типе "for(var1;2;3)4" имеет тот же смысл, что и в типе "switch". Он хранит список переменных, объявляемых в атрибуте 1. Смысл остальных аргументов данных типов легко определяется их положением в имени типа.

Пятую группу составляют вызовы функций. Они моделируются следующим типом:

```
(mot "1(2)" :at 1 "expression" :at 2 (listt "expression"))
```

Экземпляры этого типа связаны только с онтологическими моделями операторов `return`.

6. Модели агентов и окружения

Поскольку у нас последовательное вычисление и вычислитель только один, тип для окружения не имеет атрибутов:

```
(mot "env")
```

Агент для языка Си хранит достаточно много информации, но для нашего подмножества языка Си достаточно двух атрибутов:

```
(mot "agent"
  :at "location" (cot :amap "variable" "location")
  :at "value" "c value")
```

Предопределенный атрибут `"value"` имеет тип `"c value"`, который строится как объединение всех типов значений языка Си.

Атрибут `"location"` сопоставляет переменным программы связанные с ними ячейки памяти и имеет тип `(cot :amap "variable""location")`.

Тип `"location"` ячеек памяти определяется следующим образом:

```
(mot "location" :at "value" "c value")
```

Он имеет атрибут `"value"`, который хранит значение, приписанное ячейке памяти.

7. Операционная семантика моделей операторов передачи управления

Оператор break. Операционная семантика оператора `break` (более точно его онтологической модели) задается следующим атрибутным замыканием:

```
(aclosure ac :attribute "opsem" :type "break"
  :p (pop-aclosure ac) ac1 :nmatch
  :v (null ac1) T :exit (error ac "break")
  :do (match
    :ap ac1 "stage" st :do (nmatch
      :v (equal st "exiting 1(2)")
```

```
7       :exit (error ac "break")
8       :v (equal st "exiting while(1)2")
9       :v (equal st "exiting do1while(2)")
10      :v (equal st "exiting for(1;2;3)4")
11      :exit
12      :v (equal st "exiting for(var1;2;3)4")
13      :v (equal st "exiting switch(1)2")
14      :exit (eval-aclosure ac1)
15      :v (equal st "exiting {1}")
16      :exit (push-aclosure ac) (eval-aclosure ac1)
17      :do (eval-aclosure ac))))
```

Тело этого атрибутного замыкания моделирует передачу управления, осуществляемую оператором `break`, через другие операторы.

Строка `2` сохраняет в параметре `ac1` верхний элемент стека, связанного с текущим агентом (`aget ac "agent"`), удаляя этот элемент из стека в текущем окружении (`aget ac "env"`).

В строке `3` выполняется проверка, а не пуст ли этот стек (пустота стека соответствует значению `nil` параметра `ac1`). Если стек пуст, то выдается ошибка, так как это означает, что не встретился оператор, который должен был поймать переход по оператору `break`.

В строке `5` в параметре `st` сохраняется значение текущей стадии вычисления атрибутного замыкания `ac1`.

В строке `6` проверяется, является ли эта стадия стадией выхода `"exiting 1(2)"` из вычисления вызова функции. Если является, то выдается ошибка, так как такая ситуация тоже невозможна. Заметим, что единственное знание о других операторах и выражениях, которым владеет оператор передачи управления (в данном случае оператор `break`) – это знание имен стадий этих операторов, которые реагируют на передачу управления.

В строках `8-10` проверяется, является ли эта стадия стадией выхода `"exiting while(1)2"`, `"exiting do1while(2)"` и `"exiting for(1;2;3)4"` из операторов `while`, `do-while` и `for` без декларации переменных, соответственно. Если является, то передача управления завершается в строке `11`. А поскольку эти стадии являются признаками завершения соответствующих операторов, то, в соответствии с семантикой функции

`next-aclosure` управление передается следующему замыканию из стека замыканий (в частности, если после этих циклов имеется еще оператор, то управление передается ему).

В строках 12-13 проверяется, является ли эта стадия стадией выхода `"exiting for(var1;2;3)4"` и `"exiting switch(1)2"` из операторов `for` с декларацией переменных и `switch`, соответственно. Если является, то передача управления завершается в строке 14. Но, в отличие от предыдущих случаев, перед выходом из этих операторов выполняются действия, связанные с восстановлением старых ячеек памяти для переменных, а именно вычисляется атрибутное замыкание `ac1`.

В строке 15 проверяется, является ли эта стадия стадией выхода из блока. Если является, то передача управления продолжается после блока, что обеспечивается выражением `(push-aclosure ac)`, но перед этим также выполняются действия, связанные с восстановлением старых ячеек памяти для переменных, а именно вычисляется выражение `(eval-aclosure ac1)`. Заметим, что чтобы обеспечить такой порядок вычисления этих выражений, первое выражение откладывает вычисление `ac`, помещая его в стек. Поэтому сначала вычислится `ac1`, а потом восстановиться из стека и вычислиться `ac`.

Строка 17 соответствует любой другой стадии и любому другому оператору. В этом случае передача управления продолжается.

Оператор continue. Операционная семантика оператора `continue` задается во многом аналогичным образом:

```
1  (aclosure ac :attribute "opsem" :type "continue"
2    :p (peek-aclosure ac) ac1 :nmatch
3    :v (null ac1) T :exit (error ac "continue")
4    :do (match
5      :ap ac1 "stage" st :do (nmatch
6        :v (equal st "exiting 1(2)")
7        :exit (error ac "continue")
8        :v (equal st "exiting while(1)2")
9        :exit (update-eval-aclosure ac1 :stage "executing 1")
10       :v (equal st "exiting do1while(2)")
11       :exit (update-eval-aclosure ac1 :stage "executing 2")
12       :v (equal st "exiting for(1;2;3)4")
```

```
      :v (equal st "exiting for(var1;2;3)4")
      :exit (update-eval-aclosure ac1 :stage "executing 3")
      :v (equal st "exiting switch(1)2")
      :v (equal st "exiting {1}")
      :exit (pop-aclosure ac) (push-aclosure ac)
        (eval-aclosure ac1)
      :do (pop-aclosure ac) (eval-aclosure ac))))
```

Но при этом имеется несколько отличий для этого оператора.

Во-первых, для стадии `"exiting for(var1;2;3)4"` не нужно восстанавливать старые ячейки памяти для переменных, так как мы не выходим из оператора `for`.

Во-вторых, стадия `"exiting switch(1)2"` обрабатывается также, как и завершающая стадия для блока, поскольку передача управления продолжается.

В-третьих, для стадий, связанных с завершением операторов итерации управление передается на соответствующие стадии этих операторов, с которых их выполнение продолжается.

Оператор goto. Операционная семантика оператора `goto` задается следующим образом:

```
(aclosure ac :attribute "opsem" :type "goto1"
  :p (pop-aclosure ac) ac1 :nmatch
  :v (null ac1) T :exit (error ac "goto1")
  :do (match :ap ac1 "stage" st :do (nmatch
    :v (equal st "exiting 1(2)")
    :exit (error ac "goto")
    :v (equal st "exiting for(var1;2;3)4") T
    :v (equal st "exiting switch(1)2") T
    :exit (push-aclosure ac) (eval-aclosure ac1)
    :v (equal st "exiting {1}") T
    :exit (match :ap i 1 lab :ap ac1 "instance" i1
      :ap i1 "label position" lp
      :v (member lab (attributes lp)) T
      :do (match :ap i1 2 sts :ap (aget lp lab) k
```

```
15          :do (push-aclosure ac1)
16            (clear-update-eval-aclosure ac1
17              :stage "iteration" :av "current" (+ k 1)
18              :av "bound" (length sts)
19              :av "statements" sts))
20        :exit (push-aclosure ac) (eval-aclosure ac1))
21      :do (eval-aclosure ac))))
```

Для стадий `"exiting for(var1;2;3)4"` и `"exiting switch(1)2"` (строки 7 и 8) передача управления сопровождается восстановлением старых значений переменных.

Для стадии завершения блока (строка 10) в строке 11 в параметрах `lab` и `i1` сохраняются метка оператора `goto` и блок, до конца которого произошла передача управления, а в строке 12 в параметре `lp` сохраняется отображение меток, которые встречаются в операторе блока на верхнем уровне, в их позиции в списке операторов блока.

В строке 13 выполняется проверка, принадлежит ли метка `lab` меткам, которые встречаются в операторе блока на верхнем уровне.

Если проверка выполняется (строка 14), то параметрам `sts` и `k` присваиваются список операторов блока и позиция метки `lab` в этом списке. Затем запускается выполнение блока с оператора в `k + 1` позиции. Это делается на стадии `"iteration"` вычисления блока. Контекст вычисления атрибута `"opsem"` на этой стадии дополнительно включает атрибуты `"current"`, `"bound"` и `"statements"`, хранящие позицию вычисляемого в текущий момент оператора из списка операторов блока, число операторов в списке и сам список, соответственно.

Если проверка не выполняется (строка 20), то восстанавливаются старые ячейки памяти для переменных, определенных в блоке, блок завершается и передача управления продолжается.

Оператор return без аргумента. Операционная семантика этого оператора проще, чем предыдущих:

```
(aclosure ac :attribute "opsem" :type "return" :match
  :p (pop-aclosure ac) ac1 :v (not (null ac1)) ac1
  :do (match :ap ac1 "stage" st :do (nmatch
    :v (equal st "exiting for(var1;2;3)4") T
```

```
5       :v (equal st "exiting switch(1)2") T
6       :v (equal st "exiting {1}") T
7       :exit (push-aclosure ac) (eval-aclosure ac1)
8       :av (equal st "exiting 1(2)") T
9       :exit (eval-aclosure ac1)
10      :do (eval-aclosure ac)))
11    :exit (error ac "return"))
```

Для случаев, представленных в строках 4-6 также происходит восстановление старых ячеек памяти для переменных в строке 7 с продолжением передачи управления.

Единственный случай, когда передача управления завершается – это выход из вызова функции (строка 8).

Оператор return с аргументом. Операционная семантика для оператора `return`, возвращающего значение, разбивается на несколько стадий.

Для начальной стадии (она всегда имеет имя `nil`) имеем следующее определение:

```
1 (aclosure ac :attribute "opsem" :type "return1" :instance i :do
2   (update-push-aclosure ac "stage" "storing return value")
3   (clear-update-eval-aclosure ac "instance" (aget i 1)))
```

В строке 2 в стек окружения откладывается стадия `"storing return value"`, которая потом перехватит возвращаемое функцией значение и передаст его за пределы вызова функции.

В строке 3 запускается вычисление выражения, связанного с оператором `return` через атрибут `1`. Заметим, что если при вычислении выражение произойдет исключительная ситуация (например, будет послан сигнал в Си), то отложенная стадия `"storing return value"` не будет вычисляться, так как будет удалена из стека механизмом просачивания исключительной ситуации подобно тому, как просачиваются операторы передачи управления.

Стадия `"storing return value"` определяется следующим образом:

```
1 (aclosure ac :attribute "opsem" :type "return1"
2   :stage "storing return value" :value v :do
3   (update-eval-aclosure ac :stage "propagation"
```

```
4     :av "return value" v))
```

В строке 2 в параметре v сохраняется вычисленное значение выражения, связанного с оператором return. В строках 3-4 запускается стадия "propagation", которая просачивает это значение до выхода из вызова функции.

Стадия "propagation" определяется в основном аналогично единственной стадии оператора return без аргумента:

```
1  (aclosure ac :attribute "opsem" :type "return1"
2    :stage "propagation" :match
3    :p (pop-aclosure ac) ac1 :v (not (null ac1)) ac1
4    :do (match :ap ac1 "stage" st :do (nmatch
5      :v (equal st "exiting for(var1;2;3)4") T
6      :v (equal st "exiting switch(1)2") T
7      :v (equal st "exiting {1}") T
8      :exit (push-aclosure ac) (eval-aclosure ac1)
9      :av (equal st "exiting 1(2)") T
10     :exit
11       (update-push-aclosure ac :stage "returning value")
12       (eval-aclosure ac1)
13     :do (eval-aclosure ac)))
14   :exit (error ac "return1"))
```

Единственное отличие заключается в том, что после выхода из вызова функции (строка 12) выполняется еще одна стадия "returning value" оператора goto, которая делает значение выражения последним вычисленным значением, помещая его в атрибут value текущего агента.

Операционная семантика этой последней стадии выполнения оператора goto определяется следующим образом:

```
1  (aclosure ac :attribute "opsem" :type "return1"
2    :stage "returning value" :do (aget ac "return value"))
```

Эта стадия просто возвращает значение атрибута "return value" из контекста вычисления атрибута opsem в качестве значения этого атрибута.

8. Операционная семантика моделей операторов, связанных с операторами передачи управления

Выполнение оператора `switch` разбивается на 8 стадий, каждая из которых моделирует отдельный аспект его выполнения: сохранение контекста, вычисление управляющего выражения, сопоставление меток (несколько стадий) и восстановление окружения. Ниже приведено формальное описание этих стадий на языке ABML.

На нулевой стадии осуществляется переход к стадии `"entering switch(1)2"` (строка 2), которая перед выполнением оператора `switch` сохраняет ячейки памяти, связанные с переменными, объявляемыми в этом операторе на верхнем уровне:

```
1 (aclosure ac :attribute "opsem" :type "switch(1)2" :do
2   (update-eval-aclosure ac :stage "entering switch(1)2"))
```

Стадия `"entering switch(1)2"` моделируется следующим образом:

```
1 (aclosure ac :attribute "opsem" :type "switch(1)2"
2   :stage "entering switch(1)2" :instance i :agent a
3   :ap i "variables" vars :ap (mo) varlocs :do
4     (update-push-aclosure ac :stage "executing 1")
5     (dolist (var vars varlocs)
6       (aset varlocs var (aget a "location" var))))
```

В строке 3 выполняется присваивание параметру `vars` списка переменных, объявленных на верхнем уровне, и инициализация параметра `varlocs` пустым изменяемым объектом.

В строке 4 откладывается в стек выполнение стадии `"executing 1"`, которое вычисляет управляющее выражение оператора `switch`, хранящееся в атрибуте `1`.

После выполнения строк 5-6 параметр `varlocs` хранит отображение переменных из списка `vars` в ячейки памяти, связанные с ними до начала выполнения оператора `switch`. Значение этого параметра возвращается в качестве значения атрибутного замыкания.

Стадия `"executing 1"` определяется следующим образом:

```
1 (aclosure ac :attribute "opsem" :type "switch(1)2"
2   :stage "executing 1" :instance i :value v :do
3     (update-push-aclosure ac
4       :stage "exiting switch(1)2"
```

```
5       :av "variable context" v)
6     (update-push-aclosure ac :stage "executing 2")
7     (update-eval-aclosure ac :instance (aget i 1)))
```

В строке 2 в параметре v сохраняется значение переменной varlocs, вычисленной на предыдущей стадии.

После выполнения строк 3-5 в стек окружения сохраняется стадия "exiting switch(1)2", которая восстанавливает старые ячейки памяти, связанные с переменными, при завершении выполнения оператора switch.

В строке 6 в стек сохраняется стадия "exiting switch(1)2", отвечающая за выполнения тела оператора switch.

В строке 7 вычисляется управляющее выражение оператора switch.

Операционная семантика стадии "executing 2" определяется следующим образом:

```
1 (aclosure ac :attribute "opsem" :type "switch(1)2"
2   :stage "executing 2" :instance i :agent a :value v
3   :ap i 2 sts :do
4   (update-eval-aclosure ac :stage "nomatch"
5     :av "current" 0 :av "bound" (length sts)
6     :av "statements" sts :av "pattern" v))
```

В строке 2 в параметре v сохраняется значение управляющего выражения, вычисленного на предыдущей стадии.

В строке 2 в параметре sts сохраняется список операторов, составляющих тело оператора switch.

Строки 4-6 запускают выполнение стадии "nomatch", которая осуществляет последовательный просмотр операторов списка в случае, если еще не найдено сопоставление с case-меткой. Эта стадия использует параметры "current", "bound", "statements" и "pattern", хранящие позицию текущего оператора, число операторов в списке, список операторов и вычисленное значение управляющего выражения.

Стадия "nomatch" определяется следующим образом:

```
1 (aclosure ac :attribute "opsem" :type "switch(1)2"
2   :stage "nomatch" :ap ac "current" j :ap ac "bound" k
3   :ap ac "statements" sts :ap "pattern" p :match
```

```
4   :v (< j k) T :p (nth j sts) st :do
5     (nmatch
6       :v (is-instance st "case1") T
7       :exit (match :v (= (aget st 1) p) T
8         :do (update-eval-aclosure ac :stage "match"
9           :av "current" (+ j 1))
10        :exit (update-eval-aclosure ac
11          :av "current" (+ j 1)))
12      :v (is-instance st "default") T
13      :exit (update-eval-aclosure ac :stage "match"
14        :av "current" (+ j 1))
15      :do (update-eval-aclosure ac :av "current" (+ j 1))))
```

В строке 4 выполняется проверка перебраны ли все операторы из списка. Если да, то стадия завершается. Также в этой строке параметру `st` присваивается текущий оператор в качестве значения.

В строке 6 рассмотрен случай, когда текущий оператор является `case`-оператором. В этом случае в строке 7 выполняется проверка, совпадает ли метка `case`-оператора со значением управляющего выражения. Если совпадает, то в строках 8-11 выполняется переход к стадии `"match"`, которая продолжает проход по операторам списка, зная, что сопоставление уже произошло.

В строке 12 рассмотрен случай, когда текущий оператор является `default`-оператором. В этом случае также выполняется переход к стадии `"match"`.

В строке 15 рассмотрен случай, когда текущий оператор не является ни `case`-оператором, ни `default`-оператором.

Стадия `"nomatch"` задается следующим атрибутным замыканием:

```
1 (aclosure ac :attribute "opsem" :type "switch(1)2"
2   :stage "match" :ap ac "current" j :ap ac "bound" k
3   :ap ac "statements" sts :match
4   :v (< j k) :do
5     (update-push-aclosure ac :av "current" (+ j 1))
6     (clear-update-eval-aclosure ac :instance (nth j sts)))
```

В строке 6 выполняется текущий оператор, а в строке 5 осуществляется переход к следующему оператору.

И последняя стадия "exiting switch(1)2" определяется следующим образом:

```
1 (aclosure ac :attribute "opsem" :type "switch(1)2"
2   :stage "exiting switch(1)2" :agent a
3   :ap "variable context" varlocs :do
4     (dolist (var (attributes varlocs))
5       (aset a "location" var (aget varlocs var))))
```

В строке 3 в параметре `varlocs` сохраняется старое отображение переменных в связанные с ними ячейки памяти.

При выполнении строк 4-5 эти старые связи переменных и ячеек памяти становятся актуальными и сохраняются в текущем агенте.

Заметим, что при определении операционной семантики оператора `switch` рассматривалось только нормальное последовательное выполнение этого оператора. Это справедливо и для других операторов. Этого достаточно, поскольку распространение (propagation) и обработка исключительных ситуаций обеспечивается операторами, связанными с порождением таких ситуаций (операторами передачи управления, операторами порождения исключений и т. п.).

9. Родственные работы

Исследования в области формальной семантики языков программирования имеют длительную историю и охватывают широкий спектр подходов – от классических структурной операционной семантики и операционной семантики малого шага до денотационных, аксиоматических и гибридных формализаций [26, 45]. В последние годы наблюдается устойчивый интерес к развитию формальных моделей семантики промышленных языков программирования, прежде всего языка C и C-подобных языков, что обусловлено их широким применением в системном программировании и критически важных программных компонентах [18, 21, 36].

Операционная семантика языка C и его подмножеств. Язык C традиционно рассматривается как один из наиболее сложных объектов для формальной семантики из-за низкоуровневой модели памяти и неопределённого поведения. Формальные семантики C и

его подмножеств (например, Clight в CompCert) стали предметом активного исследования, как в классических работах по механизации семантики [12], так и в современных обзорах и улучшениях семантических моделей [47, 60]. Верифицированный компилятор CompCert является краеугольным камнем исследований корректности компиляции C-программ на промышленном уровне [18, 34] и продолжает развиваться [32, 54]. Дополнительные проекты, такие как Checked C, предлагают более безопасные расширения C с формальными семантическими моделями [1, 14, 15, 22, 35]. Семантики C в виде операционных семейств используются для доказательства корректности оптимизаций и анализа поведения в случае неопределенного поведения [48, 55].

Формальные семантики C-подобных языков. Помимо C, значительное число исследований посвящено семантике C-подобных языков. Например, для Rust разработаны исчерпывающие операционные семантики, включая формальные модели владения и заимствования [5, 27, 59]. LLVM IR как объединенное промежуточное представление также изучается с точки зрения формальных семантик [10, 23, 33, 37, 61, 61], что важно для семантики оптимизаций и трансформаций.

Онтологические и ориентированные на знания подходы к семантике. Использование онтологий и моделей знания становится всё более распространённым в семантических исследованиях. Онтологии формализуют концепты, отношения и правила в предметной области, что улучшает семантическую интерпретацию сложных систем [19, 24]. Хотя большинство работ по онтологиям фокусируются на семантике естественных языков или интеграции данных, методы онтологического моделирования также применялись для описания программных систем и представления семантики языков программирования [2, 41, 42].

Современные подходы к формальным семантикам взаимодействуют с логикой и построением онтологий [25, 52], обеспечивая основу для семантической интеграции и доказательств верификации.

Некоторые подходы используют методы онтологий совместно с логическими выводами и переносом знаний для динамических систем, что близко к концепции онтологической семантики исполнения [3, 7, 8, 28].

Атрибутные грамматики и расширенные модели исполнения Исследования атрибутных грамматик расширили классические синтаксические грамматики, включая семантические атрибуты и контекстно-чувствительные преобразования [16, 40, 50]. Эти модели стали основой для дескриптивных семантик, где семантика операций определяется не только формальными правилами, но и дополнительной структурной информацией о контексте исполнения [13, 40, 50].

Атрибутные грамматики остаются важным инструментом для определения семантики, особенно в расширенных моделях исполнения, где семантика контекста учитывается более гибко [6, 29, 55].

Современные расширения включают поддержку динамического контекста, агентных систем и взаимодействия с внешней средой [13, 51, 53].

Связь с задачами анализа и верификации программ. Формальные семантики активно используются в статическом анализе, доказательстве корректности и проверке оптимизаций [4, 9, 11, 17, 31, 38, 39, 43, 49, 57, 58, 62].

Семантика LLVM IR, а также моделей C и Rust, обеспечивает основу для автоматической верификации и анализа системного программного обеспечения [10, 20, 56].

Онтологическая операционная семантика обещает более лёгкую интеграцию с системами логического вывода, поскольку онтологии являются стандартным формализмом для представления знаний и могут напрямую связываться с автоматизированными анализаторами [30, 44, 46].

Сравнение с настоящей работой. В отличие от большинства исследований, где семантика операторов передачи управления задаётся в рамках фиксированной модели состояний, настоящая работа предлагает интерпретацию операционной семантики как динамики онтологических моделей, обеспечивая модульность и концептуальную целостность спецификации. Это позволяет естественно выразить нелокальную передачу управления и восстановление контекста исполнения, что остаётся сложной задачей для традиционных подходов.

10. Заключение

В данной работе был предложен онтологический подход к заданию операционной семантики операторов передачи управления языка C с использованием предметно-ориенти-

рованного языка ABML. Показано, что системы переходов, традиционно применяемые для определения операционной семантики, естественным образом интерпретируются как дискретные динамические системы и могут быть формализованы в терминах онтологического моделирования.

В рамках работы была построена онтология операторов передачи управления, охватывающая операторы goto, break, continue и return, а также онтологии конструкций, реагирующих на передачу управления, включая помеченные операторы, блоки и оператор switch. Для этих онтологических моделей была задана операционная семантика в виде атрибутных замыканий относительно атрибута opsem, вычисляемых в контексте агентов и окружения.

Существенным результатом является адаптация языка ABML к задачам задания семантики языков программирования. Уточнение понятия атрибутного замыкания, введение стадий вычисления и явное моделирование контекста выполнения позволили выразить сложные аспекты семантики, такие как нелокальная передача управления и восстановление состояния после завершения операторов.

Предложенный подход обладает рядом преимуществ по сравнению с традиционными формализациями. Он обеспечивает модульность спецификаций, возможность повторного использования онтологических моделей, а также естественную расширяемость при добавлении новых конструкций языка. Кроме того, онтологическое представление создает предпосылки для интеграции с инструментами анализа знаний, логического вывода и верификации.

В дальнейшем представляется перспективным расширение разработанной модели на другие конструкции языка C, включая выражения, функции и механизмы обработки исключений, а также применение подхода к другим языкам программирования. Отдельный интерес представляет использование онтологической операционной семантики для анализа свойств программ, построения интерпретаторов и разработки формальных средств верификации.

Список литературы

1. Achieving safety incrementally with Checked C / Ruef A., Lampropoulos L., Sweet I., Tarditi D., and Hicks M. // International Conference on Principles of Security and Trust / Springer International Publishing Cham. — 2019. — P. 76–98.

2. Adamo G., Villa F. Ontology of Descriptions and Observations for Integrated Modelling (ODO-IM). — 2024.
3. Angele J., Kifer M., Lausen G. Ontologies in F-logic // Handbook on ontologies. — Springer, 2009. — P. 45–70.
4. Appel A. W. Program logics for certified compilers. — Cambridge University Press, 2014.
5. Atkey R. e. a. Semantic Foundations for Rust Ownership // Journal of Functional Programming. — 2022.
6. Author A. Advanced Attribute Grammars: Contextual Semantics // Journal of Formal Languages. — 2021.
7. Babaei Giglou H., D'Souza J., Auer S. LLMs4OL: Large language models for ontology learning // International Semantic Web Conference / Springer. — 2023. — P. 408–427.
8. Balaban M. The F-logic approach for description languages // Annals of Mathematics and Artificial Intelligence. — 1995. — Vol. 15, no. 1. — P. 19–60.
9. Barroso P., Pereira M., Ravara A. Leroy and Blazy Were Right: Their Memory Model Soundness Proof // Verified Software. Theories, Tools and Experiments.: 14th International Conference, VSTTE 2022, Trento, Italy, October 17–18, 2022, Revised Selected Papers / Springer Nature. — 2023. — Vol. 13800. — P. 20.
10. Beck C., Chen H., Zdancewic S. Vellvm: Formalizing the Informal LLVM: (Experience Report) // NASA Formal Methods Symposium / Springer. — 2025. — P. 91–99.
11. Beringer L., Appel A. W. Abstraction and subsumption in modular verification of C programs // Formal Methods in System Design. — 2021. — Vol. 58, no. 1. — P. 322–345.
12. Blazy S., Leroy X. Mechanized Semantics for the Clight Subset of the C Language // Journal of Automated Reasoning. — 2009. — Vol. 43. — P. 263–288.
13. Bridging MDE and AI: a systematic review of domain-specific languages and model-driven practices in AI software systems engineering / Rädler S., Berardinelli L., Winter K., Rahimi A., and Rinderle-Ma S. // Software and Systems Modeling. — 2025. — Vol. 24, no. 2. — P. 445–469.
14. Checked C: Making C safe by extension / Elliott A. S., Ruef A., Hicks M., and Tarditi D. // 2018 IEEE Cybersecurity Development (SecDev) / IEEE. — 2018. — P. 53–60.
15. Checkedcbox: Type directed program partitioning with checked c for incremental spatial memory safety / Li L., Bhattar A., Chang L., Zhu M., and Machiry A. // arXiv preprint arXiv:2302.01811. — 2023.

16. Ciccaglione M., Caliandro P., Pellegrini A. Tahr: The Generative Attribute Grammar Framework // arXiv preprint arXiv:2512.01872. — 2025.
17. Cohen J. M., Wang Q., Appel A. W. Verified erasure correction in Coq with MathComp and VST // International Conference on Computer Aided Verification / Springer. — 2022. — P. 272–292.
18. CompCert: A Formally Verified C Compiler. — `https://compcert.org`. — 2025.
19. The computer science ontology: A comprehensive automatically-generated taxonomy of research areas / Salatino A. A., Thanapalasingam T., Mannocci A., Birukou A., Osborne F., and Motta E. // Data Intelligence. — 2020. — Vol. 2, no. 3. — P. 379–416.
20. Crux, a precise verifier for rust and other languages / Pernsteiner S., Diatchki I. S., Dockins R., Dodds M., Hendrix J., Ravich T., Redmond P., Scott R., and Tomb A. // arXiv preprint arXiv:2410.18280. — 2024.
21. A Formal Semantics of C with Applications : Rep. / Technical Report ; executor: Ellison C., Roșu G. : 2010. — Access mode: `https://iris-project.org/pdfs/2025-icfp-osiris.pdf`.
22. A formal model of Checked C / Li L., Liu Y., Postol D., Lampropoulos L., Van Horn D., and Hicks M. // Journal of Computer Security. — 2023. — Vol. 31, no. 5. — P. 581–614.
23. Formalizing the LLVM intermediate representation for verified program transformations / Zhao J., Nagarakatte S., Martin M. M., and Zdancewic S. // Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. — 2012. — P. 427–440.
24. Guizzardi G., Guarino N. Explanation, semantics, and ontology // Data & Knowledge Engineering. — 2024. — Vol. 153. — P. 102325.
25. Harper R. Semantic Foundations for Programming Languages. — MIT Press, 2023.
26. Hoare C. A. R., He J. Unifying Theories of Programming. — Prentice Hall, 1998.
27. Kan S. e. a. An Executable Operational Semantics for Rust // arXiv preprint arXiv:1804.07608. — 2018. — Access mode: `https://arxiv.org/abs/1804.07608`.
28. Kifer M., Lausen G., Wu J. Logical foundations of object-oriented and frame-based languages // Journal of the ACM (JACM). — 1995. — Vol. 42, no. 4. — P. 741–843.
29. Knuth D. E. Semantics of Context-Free Languages // Mathematical Systems Theory. — 1968.
30. Koopmann P. Explaining Reasoning Results for Description Logic Ontologies // Joint

Proceedings of the 20th and 21st Reasoning Web Summer Schools (RW 2024 & RW 2025) / Schloss Dagstuhl–Leibniz-Zentrum für Informatik. — 2025. — P. 6–1.

31. Krebbers R. A formal C memory model for separation logic // Journal of Automated Reasoning. — 2016. — Vol. 57, no. 4. — P. 319–387.
32. Krebbers R., Leroy X., Wiedijk F. Formal C semantics: CompCert and the C standard // International Conference on Interactive Theorem Proving / Springer. — 2014. — P. 543–548.
33. Lee J. A Validated Semantics for LLVM IR. — PhD thesis, IST UT Lisbon. — 2024. — Access mode: `https://web.ist.utl.pt/nuno.lopes/students/Juneyoung_Lee_PhD.pdf`.
34. Leroy X. Formal certification of a compiler back-end or: programming a compiler with a proof assistant // Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — 2006. — P. 42–54.
35. Leroy X. Formal verification of a realistic compiler // Communications of the ACM. — 2009. — Vol. 52, no. 7. — P. 107–115.
36. Leroy X., Blazy S. From Mechanized Semantics to Verified Compilation: the Clight Semantics of CompCert // FASE 2024. — Springer. — 2024.
37. Li L., Gunter E. L. K-LLVM: a relatively complete semantics of LLVM IR // 34th European Conference on Object-Oriented Programming (ECOOP 2020) / Schloss Dagstuhl–Leibniz-Zentrum für Informatik. — 2020. — P. 7–1.
38. Mansky W. Bringing Iris into the Verified Software Toolchain // arXiv preprint arXiv:2207.06574. — 2022.
39. Mansky W., Du K. An Iris instance for verifying CompCert C programs // Proceedings of the ACM on Programming Languages. — 2024. — Vol. 8, no. POPL. — P. 148–174.
40. Michaelson D., Nadathur G., Van Wyk E. A modular approach to metatheoretic reasoning for extensible languages // ACM Transactions on Programming Languages and Systems. — 2025. — Vol. 47, no. 3. — P. 1–56.
41. Na Nongkhai L., Wang J., Mendori T. Development and evaluation of adaptive learning support system based on ontology of multiple programming languages // Education Sciences. — 2025. — Vol. 15, no. 6. — P. 724.
42. Nongkhai L. N., Wang J., Mendori T. Developing an Ontology of Multiple Programming Languages from the Perspective of Computational Thinking Education. // International Association for Development of the Information Society. — 2022.
43. Park S. H., Pai R., Melham T. A formal CHERI-C semantics for verification // International

Conference on Tools and Algorithms for the Construction and Analysis of Systems / Springer. — 2023. — P. 549–568.

44. Pileggi S. F. Ontology in hybrid intelligence: A concise literature review // Future Internet. — 2024. — Vol. 16, no. 8. — P. 268.
45. Plotkin G. D. A Structural Approach to Operational Semantics. — University of Edinburgh, 1970.
46. Qureshi H. M., Faber W. Evaluating Datalog Tools for Meta-reasoning over OWL 2 QL // Theory and Practice of Logic Programming. — 2024. — Vol. 24, no. 2. — P. 368–393.
47. Ramel D. Modernizing C for Security: the Checked C Initiative // ADTmag. — 2016.
48. Rasband V. e. a. Formalizing Undefined Behavior in C // ACM SIGPLAN Notices. — 2020.
49. RefinedC: automating the foundational verification of C code with refined ownership types / Sammler M., Lepigre R., Krebbers R., Memarian K., Dreyer D., and Garg D. // Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. — 2021. — P. 158–174.
50. Ringo N., Kramer L., Van Wyk E. Nanopass attribute grammars // Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering. — 2023. — P. 70–83.
51. The rise of agentic AI: A review of definitions, frameworks, architectures, applications, evaluation metrics, and challenges / Bandi A., Kongari B., Naguru R., Pasnoor S., and Vilipala S. V. // Future Internet. — 2025. — Vol. 17, no. 9. — P. 404.
52. Roșu G., Serbanuta T. K Framework and Formal Semantics // Journal of Functional Programming. — 2010.
53. A survey of ai agent protocols / Yang Y., Chai H., Song Y., Qi S., Wen M., Li N., Liao J., Hu H., Lin J., Chang G., et al. // arXiv preprint arXiv:2504.16736. — 2025.
54. Thibault J. e. a. SECOMP: Formally Secure Compilation of Compartmentalized C Programs // arXiv preprint arXiv:2401.16277. — 2024. — Access mode: `https://arxiv.org/abs/2401.16277`.
55. Towler C. e. a. Operational Semantics for Low-Level Languages // Journal of Programming Languages. — 2023.
56. Van Oorschot D., Huisman M., Şakar Ö. First steps towards deductive verification of LLVM IR // International Conference on Fundamental Approaches to Software Engineering / Springer. — 2024. — P. 290–303.

57. Vst-a: A foundationally sound annotation verifier / Zhou L., Qin J., Wang Q., Appel A. W., and Cao Q. // Proceedings of the ACM on Programming Languages. — 2024. — Vol. 8, no. POPL. — P. 2069–2098.
58. VST-Floyd: A separation logic tool to verify correctness of C programs / Cao Q., Beringer L., Gruetter S., Dodds J., and Appel A. W. // Journal of Automated Reasoning. — 2018. — Vol. 61, no. 1. — P. 367–422.
59. Wang F. e. a. KRust: A Formal Executable Semantics of Rust // arXiv preprint arXiv:1804.10806. — 2018. — Access mode: https://arxiv.org/abs/1804.10806.
60. Wils S., Jacobs B. Certifying C Program Correctness with Respect to CompCert with VeriFast // arXiv preprint arXiv:2110.11034. — 2021. — Access mode: https://arxiv.org/abs/2110.11034.
61. Zakowski Y. e. a. Modular, Compositional, and Executable Formal Semantics for LLVM IR. — 2021. — Access mode: https://dl.acm.org/doi/10.1145/3473572.
62. Zhao Y., Sanan D. Rely-guarantee Reasoning about Concurrent Memory Management: Correctness, Safety and Security // arXiv preprint arXiv:2309.09997. — 2023.
63. Ануреев И.С. Язык спецификации дискретных динамических систем, ориентированных на знания, структурированные в онтологиях // Системная информатика. — 2025. — no. 29. — P. 137–158.